Графы: DFS. Мосты. Точки сочленения.

Сапожников Денис

Contents

1 Определения

Определение (Граф)**.** Граф $G(V, E)$ — это множество вершин V и множество рёбер E — множество пар вершин $(a, b) : a, b \in V$, которые "соединены".

Определение (Подграф)**.** Подграф $G'(V, E') \subset G(V, E)$ — это граф на том же множестве вершин V, и подмножестве рёбер $E' \subseteq E$.

Везде ниже мы будем считать, что $|V| = n, |E| = m$.

Определение (Петля)**.** Ребро $(a, b) \in E$ называется петлёй, если оно соединяет вершину с самой собой, т.е. $a = b$.

Определение (Кратное ребро)**.** Два ребра называются кратными, если они соединяют две одинаковые пары рёбер.

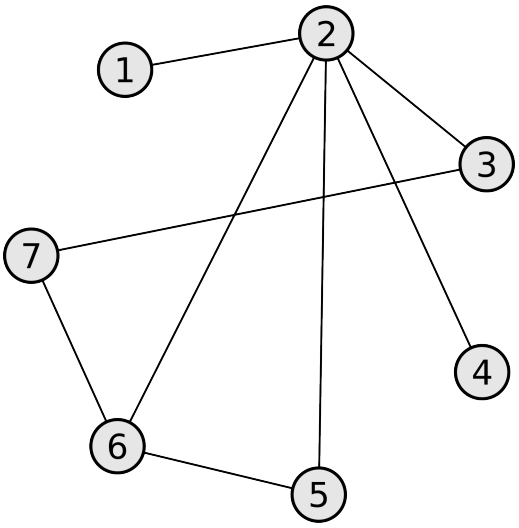
Как правило в задачах указано, что нет петель и кратных рёбер, но если такое не написано, то с ними стоит быть осторожнее!

Определение (Взвешенный граф)**.** Граф называется взвешенным, если есть функция $w : E \to \mathbb{R}$, называемая весом ребра; иначе говоря, каждому ребру сопоставляется вещественное число.

Определение (Дерево)**.** Дерево — это связный граф на n вершинах и $n - 1$ ребре.

2 Хранение графов

Чтобы решать задачки на графы, вы рисуете их на бумажке в виде вершин и рёбер.



Но как же сохранить графы в памяти? Существуют два подхода.

2.1 Матрица смежности

Вы можете сознать матрицу $n \times n$, состоящую из 0 и 1, где 1 в позиции (i, j) обозначает наличие ребра из i-й вершины в j-ю. У данного подхода есть масса преимуществ:

1. Простота. Действительно, заполнить матрицу очень просто, а хранить вам нужно лишь двумерный массив.
2. Легко проверять наличие ребра между любыми двумя вершинами.
3. Легко делать граф ориентированным/неориентированным, взвешенным/не взвешенным (для взвешенного графа можно хранить не 1 при наличии ребра, а вес ребра между вершинами).

Например, хранение неориентированного взвешенного графа будет следующим:

```
int n, m; // vertexes, edges
cin >> n >> m;
vector<vector<int>> adj(n, vector<int>(n));
for (int i = 0; i < m; ++i) {
  int a, b, w;
  cin >> a >> b >> w; // weight of edge between a and b is w
  adj[a - 1][b - 1] = adj[b - 1][a - 1] = w;
}
```

Однако, есть очень большой недостаток: если в графе 10^5 вершин и 10^5 рёбер, то вам придется сохранить таблицу размера $10^5 \times 10^5$, при этом единицы в такой таблице будет очень мало. Такие графы называются разреженными и очень часто в задачах даны именно разреженные графы. Как хранить разреженные графы?

2.2 Список смежности

Вместо того, чтобы хранить всю матрицу смежности давайте для каждой вершины хранить список её соседей — список смежности.

Такой подход, очевидно, занимает $O(n + m)$ памяти, где n – количество вершин, m – количество рёбер.

Но есть и пара проблем:

1. Неудобно проверять наличие ребра между парой вершин. Для этого придется хранить не список смежных вершин, а множество смежных вершин, что увеличивает асимптотику. Благо, в задачах почти никогда не надо проверять наличие ребра между конкретными двумя вершинами.

2. Не очень удобно хранить веса рёбер: вместе с соседом вершины придется хранить ещё и вес ребра (то есть хранить пару).

То есть теперь для хранения неориентированного взвешенного графа вам придется написать следующий код:

```
int n, m;
cin >> n >> m;
vector<vector<pair<int, int>>> gr(n); // from - { {to[1], w[1]}, ...}
for (int i = 0; i < m; ++i) {
  int a, b, w;
  cin >> a >> b >> w;
  --a, --b;
  gr[a].push_back({ b, w });
  gr[b].push_back({ a, w });
}
```

3 DFS

3.1 Алгоритм

Скорее всего, все уже знакомы с этим алгоритмом обхода графа. Напомню, что этот алгоритм «идёт, пока может», то есть:

```
bool used[N];
void dfs(int v) {
  used[v] = true;
  for (int u : gr[v])
    if (!used[u])
      dfs(u);
}
```

3.2 Красим в три цвета

Казалось бы, говорить об этих 7 строках кода нечего, но на самом деле тут есть потаенный смысл. Пусть ещё непосещённые вершины будут белыми, серыми те, которые лежат в стеке вызова, а чёрные — те, которые мы посетили и удалили из стека.

То есть:

```
enum { WHITE, GREY, BLACK };
int used[N];
void dfs(int v) {
  used[v] = GREY;
    for (int u : gr[v])
      if (used[u] == WHITE)
        dfs(u);
  used[v] = BLACK;
}
```

Лемма 1. *Не существует такого момента выполнения поиска в глубину, в который бы существовало ребро из чёрной вершины в белую.*

Proof. Пусть в процессе выполнения процедуры dfs нашлось ребро из чёрной вершины v в белую вершину u. Рассмотрим момент времени, когда мы запустили $dfs(v)$. В этот момент вершина v была перекрашена из белого в серый, а вершина u была белая. Далее в ходе выполнения алгоритма будет запущен $dfs(u)$, поскольку обход в глубину обязан посетить все белые вершины, в которые есть ребро из v. По алгоритму вершина v будет покрашена в чёрный цвет тогда, когда завершится обход всех вершин, достижимых из неё по одному ребру, кроме тех, что были рассмотрены раньше неё. Таким образом, вершина v может стать чёрной только тогда, когда dfs выйдет из вершины u, и она будет покрашена в чёрный цвет. Получаем противоречие. □

3.3 Лемма о белом пути

Лемма 2 (о белом пути). *Пусть дан граф G. Запустим $dfs(G)$. Остановим выполнение процедуры dfs от какой-то вершины u графа G в тот момент, когда вершина u была выкрашена в серый цвет (назовём его первым моментом времени). Заметим, что в данный момент в графе G есть как белые, так и чёрные, и серые вершины. Продолжим выполнение процедуры $dfs(u)$ до того момента, когда вершина u станет чёрной (второй момент времени). Тогда*

вершины графа $G \setminus u$, бывшие чёрными и серыми в первый момент времени, не поменяют свой цвет ко второму моменту времени, а белые вершины либо останутся белыми, либо станут чёрными, причём чёрными станут те, что были достижимы от вершины u по белым путям.

Proof. Чёрные вершины останутся чёрными, потому что цвет может меняться только по схеме белый $\to$ серый $\to$ чёрный. Серые останутся серыми, потому что они лежат в стеке рекурсии и там и останутся.

Далее докажем два факта:

Утверждение. *Если вершина была достижима по белому пути в первый момент времени, то она стала чёрной ко второму моменту времени.*

Proof. Если вершина v была достижима по белому пути из u, но осталась белой, это значит, что во второй момент времени на пути из u в v встретится ребро из черной вершины в белую, чего не может быть по лемме, доказанной выше. □

Утверждение. *Если вершина стала чёрной ко второму моменту времени, то она была достижима по белому пути в первый момент времени.*

Proof. Рассмотрим момент, когда вершина v стала чёрной: в этот момент существует серый путь из u в v, а это значит, что в первый момент времени существовал белый путь из u в v, что и требовалось доказать. □

Отсюда следует, что если вершина была перекрашена из белой в чёрную, то она была достижима по белому пути, и что если вершина как была, так и осталась белой, она не была достижима по белому пути, что и требовалось доказать. □

3.4 Поиск цикла

Утверждение. *В ориентированном графе существует цикл тогда и только тогда, когда при обходе dfs-ом найдется момент времени, когда мы посмотрим из серой вершины в серую.*

```
bool has_cycle(int v) { // return true if has cycle
  used[v] = GRAY;
  for (int u : gr[v]) {
    if (used[u] == GRAY || used[u] == WHITE && dfs(u)) {
      return true;
    }
  }
  used[v] = BLACK;
  return false;
}
```

Подумайте, как можно восстановить этот цикл.

3.5 Прямые и обратные рёбра

Определение. Назовём ребро **прямым**, если мы прошли по нему во время обхода dfs.

Определение. Прямые ребра образуют **дерево обхода** dfs.

Определение. Ребра (u, v), соединяющие вершину u с её предком v в дереве обхода в глубину назовём **обратными рёбрами** (для неориентированного графа предок должен быть не родителем, так как иначе ребро будет являться ребром дерева).

Определение. Все остальные ребра назовём **перекрёстными рёбрами**.

Задача. Докажите, что при обходе неориентированного графа в глубину **не существует перекрёстных рёбер**.

4 Мосты и точки сочленения

4.1 Мосты

Определение. Ребро в графе будет называться **мостом**, если при удалении его, граф распадётся на 2 компоненты связности.

Лемма. *Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины v. Тогда, если текущее ребро (v, u) таково, что из вершины u и из любого её потомка в дереве обхода в глубину нет обратного ребра в вершину v или какого-либо её предка, то это ребро является мостом. В противном случае оно мостом не является.*

Proof. В самом деле, мы этим условием проверяем, нет ли другого пути из v в u, кроме как спуск по ребру (v, u) дерева обхода в глубину. □

Теперь осталось научиться проверять этот факт для каждой вершины эффективно. Для этого воспользуемся «временами входа в вершину», вычисляемыми алгоритмом поиска в глубину.

Итак, пусть h_v — это глубина вершины в дереве dfs. Теперь введём массив fup_v, который и позволит нам отвечать на вышеописанные запросы. Время fup_v равно глубине самой высокой вершины, в которую мы можем попасть из v или её поддерева, более формально это можно записать так:

$$fup_v = \min \begin{cases} h_v & \\ h_p, & (v, p) - \text{это обратное ребро} \\ fup_u, & (v, u) - \text{это прямое ребро} \end{cases}$$

Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын u, что $fup_u \le h_v$. (Если $fup_u = h_v$, то это означает, что найдётся обратное ребро, приходящее точно в v; если же $fup_u < h_v$, то это означает наличие обратного ребра в какого-либо предка вершины v.)

Таким образом, если для текущего ребра (v, u) (принадлежащего дереву поиска) выполняется $fup_u > h_v$, то это ребро является мостом; в противном случае оно мостом не является.

```
vector<pair<int, int>> gr[N];
bool used[N];
int h[N], fup[N];
vector<int> bridges;

void dfs(int v, int p_id = -1) {
  used[v] = true;
  fup[v] = h[v];

  for (auto [u, id] : gr[v]) {
    if (id != p_id) {
      if (used[u]) {
        fup[v] = min(fup[v], h[u]);
      }
      else {
        h[u] = h[v] + 1;
        dfs(u, id);
        fup[v] = min(fup[v], fup[u]);

        if (fup[u] > h[v]) {
          bridges.push_back(id);
        }
      }
    }
  }
}
```

Стоит заметить, то если в графе нет кратных рёбер, то реализация будет проще и не нужно будет хранить номера рёбер.

4.2 Точки сочленения

Определение. Вершина в графе будет называться **точкой сочленения**, если при удалении её, граф распадётся на 2 компоненты связности.

На самом деле это почти то же самое, что и мосты. Оставаясь в той же терминологии, что и в мостах, вершина будет точкой сочленения, если $fup_u \geq h_v$ для хоты бы одного сына u вершины v. Таким образом, код нужно поменять лишь в одном месте для всех вершин, кроме корня. Корень – это отдельный случай, он будет являться точкой сочленения, если из него мы запустимся хотя бы в 2 сына (не путать со степенью вершины!).

```
vector<pair<int, int>> gr[N];
bool used[N];
int h[N], fup[N];
bool is_ap[N];

void dfs(int v, int p_id = -1) {
  used[v] = true;
  fup[v] = h[v];

  int cnt = 0;

  for (auto [u, id] : gr[v]) {
    if (id != p_id) {
      if (used[u]) {
        fup[v] = min(fup[v], h[u]);
      }
      else {
        h[u] = h[v] + 1;
        dfs(u, id);
        ++cnt;
        fup[v] = min(fup[v], fup[u]);

        if (p_id != -1 && fup[u] >= h[v]) {
          is_ap[v] = true;
        }
      }
    }
  }

  if (cnt > 1 && p_id == -1) {
    is_ap[v] = true;
  }
}
```

Забавный факт из теории графов: в графе нет точек сочленения тогда и только тогда, когда граф можно получить следующим алгоритмом:

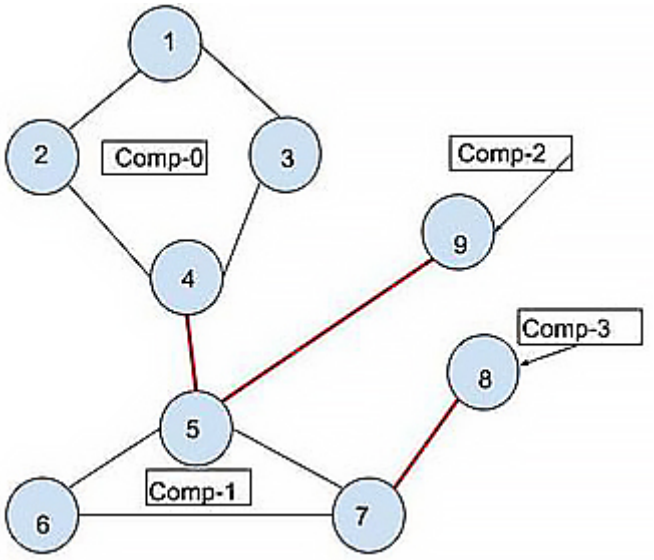
1. Создать простой цикл
2. Соединить две существующие вершины простым путём
3. Соединить две существующие вершины простым путём
4. и т.д.

В целом, такое разбиение графа на пути называется ушной декомпозицией.

5 Компоненты реберной двусвязности

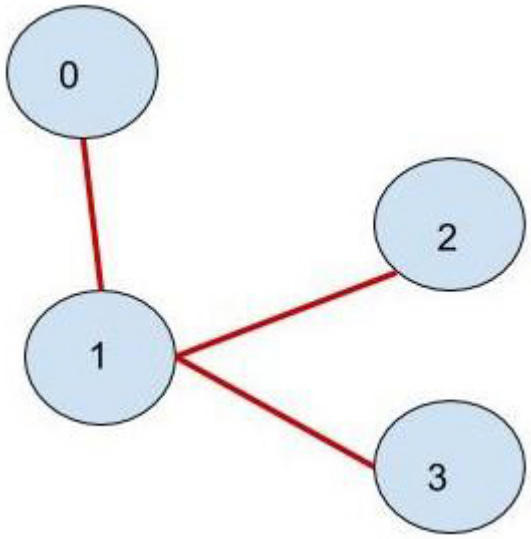
5.1 Немного теории

Мы будем говорить, что две вершины a и b лежат в одной компоненте реберной двусвязности, если существует два рёберно-непересекающихся пути из одной вершины в другую. Нарисовав пару примеров, можно заметить, что эти множества выглядят как компоненты, которые разделены мостами.



И действительно, если есть две вершины, которые разделены мостом, то они точно лежат в разных компонентах реберной двусвязности. Если же вершины лежат в одной компоененте реберной двусвязности, то можно (не очень сложно) доказать, что найдется два реберно-непересекающихся пути между этими вершинами. Для этого нужно опять же построить дерево dfs и придумать конструктивно как строить эти два пути.

А далее замечаем, что можно сжать компоненты рёберной двусвязности в одну вершину и получить «сжатый» граф (англ. bridge-block tree). Например, пример выше превратится в следующий граф:



И не случайно полученный граф является деревом! Так будет всегда, оставлю это в качестве простого упражнения.

5.2 Зачем нам это?

Часто, когда задача на дереве, она становится сильно проще. Вот и здесь, мы упрощаем структуру графа, что позволяет думать о задаче немного в другой структуре. Разберемся на примерах:

Задача. Дан неориентированный связный граф. Необходимо за линейное время ориентировать рёбра так, чтобы из каждой вершины по прежнему можно было добраться во все остальные вершины или сообщить, что это невозможно.

Решение. Заметим, что если в графе есть мост, то ориентровать в таком случае рёбра не выйдет. Во всех остальных случаях достаточно ориентировать прямые ребра графа вниз, а обратные – вверх.

Задача. Дан граф дорог. Вы – человек, который живет в городе a и работает в городе b, и платите налог в 1 рубль только за «важные» дороги. Дорога называется важной, если при её удалении вы не сможете добраться до работы. Посчитайте, сколько рублей вы должны отсыпать в казну.

Решение. Ответ равен длине пути в bridge-block tree. Можно усложнить задачу, сказав, что вы ищете ответ для нескольких людей, но тогда придется знать что такое LCA.

Задача. Дан неориентированный граф. Каждый день из каждой вершины в каждую идёт по одному человеку (всего $\frac{n(n+1)}{2}$ пар вершин). Разрешается ввести налог ровно на одной дороге в один рубль. Если человек проходит по этой дороге, то он платит рубль. Очевидно, если есть способ не платить налоги, то человек выберет ровно такой маршрут. Ваша задача — максимизировать минимальную прибыль в виде налога.

Решение. Очевидно, что оптимально ставить налог в ребро-мост. Тогда посчитаем, сколько вершин по одну сторону vu-моста (l_v) и по другую r_u. Нам нужно найти максимум величины $l_v \cdot r_u$, это можно сделать за линейное время.

Задача. Вам необходимо сказать, существует ли маршрут через три вершины a, b и c такой, что вы не пройдете по одному и тому же ребру дважды.

Решение. Существует тогда и только тогда, когда вершина b лежит на пути от a до c в bridge-block tree.